La catedral y el bazar

Linux es subversivo

Título original: The Cathedral and the Bazaar: Linux is Subversive

Eric S. Raymond, 1997

Traducción: José Soto Pérez, 1998

La catedral y el bazar es un libro sobre el software de código abierto escrito por el hacktivista Eric S. Raymond (ESR) en 1997. Ha tenido dos secuelas tituladas *Colonizando la noósfera* y *El caldero mágico*. El ensayo analiza dos modelos de producción de software bien diferenciados. Por un lado, la catedral, que representa el modelo de desarrollo más hermético y vertical característico del Software propietario; y por el otro, el bazar, con su dinámica horizontal y "bulliciosa", que caracterizó al desarrollo del kernel Linux y otros proyectos de software libre que se potenciaron con el trabajo comunitario a través de Internet del código abierto.

ERIC STEVEN RAYMOND (Boston, Massachusetts, 4 de diciembre de 1957), también conocido como ESR, es un programador informático y autor norteamericano. Su compromiso con la cultura hacker empieza en 1976, y escribió su primer proyecto de código abierto en 1982. Fue el coordinador del cliente de correo electrónico Fetchmail. También colaboró con los modos de edición de Emacs y coescribió porciones de la biblioteca GNU ncurses. Ha escrito una implementación en C del lenguaje de programación de parodia INTERCAL. Tras 1997, se convirtió en uno de los principales teorizadores del movimiento del código abierto y uno de los fundadores del Open Source Initiative. Tomó el papel de embajador del código abierto en la prensa, los negocios y en la cultura de masas. Por un tiempo fue un polémico bloggero, hasta 2006 en que dejó de hacerlo, retomándolo en 2008, citando problemas judiciales como causa de su silencio. De entre sus éxitos, hace falta destacar su contribución en la liberación del código de Netscape para acontecer el proyecto Mozilla en 1998. Se le reconoce haber llevado el código abierto a los círculos de Wall Street de una forma más efectiva que anteriores partidarios. Como autor, ha publicado varios libros y manifiestos, como Una breve historia de los hackers (1992), La catedral y el bazar (1997; ensayo publicado en formato libro en 1999), Colonizando la noosfera (1999), El caldero mágico (1999), La venganza de los hackers (1999) o El arte de programar Unix (2003). Raymond es un neopagano, un confeso anarcocapitalista y un defensor del derecho a poseer y utilizar armas de fuego. Tiene un gran interés en la ciencia ficción. Es músico amateur y cinturón negro de taekwondo.

Prólogo

Analizo un exitoso proyecto de software libre (fetchmail), que fue realizado para probar deliberadamente algunas sorprendentes ideas sobre la ingeniería de software sugeridas por la historia de Linux. Discuto estas teorías en términos de dos estilos de desarrollo fundamentalmente opuestos: el modelo catedral de la mayoría de los fabricantes de software comercial contra el

modelo bazar del mundo Linux. Demuestro que estos modelos parten de puntos de vista contrapuestos acerca de la naturaleza de la tarea de depuración del software. Posteriormente, hago una argumentación, a partir de la experiencia de Linux, de la siguiente sentencia: "si se tienen las miradas suficientes, todas los bugs saltarán a la vista". Al final, sugiero algunas fructíferas analogías con otros sistemas autoregulados de agentes egoístas, y concluyo con una somera exploración de las implicaciones que pude tener este enfoque en el futuro del software.

---

1. La catedral y el bazar

Linux es subversivo. ¿Quién hubiera pensado hace apenas cinco años que un sistema operativo de talla mundial surgiría, como por arte de magia, gracias a la actividad hacker desplegada en ratos libres por varios miles de programadores diseminados en todo el planeta, conectados solamente por los tenues hilos de la Internet?

Lo que si es seguro es que yo no. Cuando Linux apareció en mi camino, a principios de 1993, yo tenía invertidos en UNIX y el desarrollo de software libre alrededor de diez años. Fui uno de los primeros en contribuir con GNU a mediados de los ochentas y he estado aportando una buena cantidad de software libre a la red, desarrollando o colaborando en varios programas (NetHack, los modos VC y GUD de Emacs, xlife y otros) que todavía son ampliamente usados. Creí que sabía cómo debían hacerse las cosas.

Linux vino a trastocar buena parte de lo que pensaba que sabía. Había estado predicando durante años el evangelio UNIX de las herramientas pequeñas, de la creación rápida de prototipos y de la programación evolutiva. Pero también creía que existía una determinada complejidad crítica, por encima de la cual se requería un enfoque más planificado y centralizado. Yo pensaba que el software de mayor envergadura (sistemas operativos y herramientas realmente grandes, tales como Emacs) requería construirse como las catedrales, es decir, que debía ser cuidadosamente elaborado por genios o pequeñas órdenes de magos trabajando encerrados a piedra y lodo, sin liberar versiones beta antes de tiempo.

El estilo de desarrollo de Linus Torvalds ("libere rápido y a menudo, delegue todo lo que pueda, sea abierto hasta el punto de la promiscuidad") me cayó de sorpresa. No se trataba de ninguna forma reverente de construir la catedral. Al contrario, la comunidad Linux se asemejaba más a un bullicioso bazar de Babel, colmado de individuos con propósitos y enfoques dispares (fielmente representados por los repositorios de archivos de Linux, que pueden aceptar aportes de quien sea), de donde surgiría un sistema estable y coherente únicamente a partir de una serie de artilugios.

El hecho de que este estilo de bazar parecía funcionar, y funcionar bien, realmente me dejó sorprendido. A medida que iba aprendiendo a moverme en ese medio, no sólo trabajé arduamente en proyectos individuales, sino en tratar de comprender por qué el mundo Linux no naufragaba en el mar de la confusión, sino que se fortalecía con una rapidez inimaginable para los constructores de catedrales.

Creí empezar a comprender a mediados de 1996. El destino me dio un medio perfecto para demostrar mi teoría, en la forma de un proyecto de software libre que trataría de realizar siguiendo el estilo del bazar de manera consciente. Así lo hice y resultó un éxito digno de consideración.

En el resto de este artículo relataré la historia de este proyecto, y la usaré para proponer algunos aforismos sobre el desarrollo real del software libre. No todas estas cosas fueron aprendidas del mundo Linux, pero veremos como fue que les vino otorgar un sentido particular. Si estoy en lo cierto, le servirán para comprender mejor qué es lo que hace a la comunidad linuxera tan buena fuente de software; y le ayudarán a ser más productivo.

---

2. El correo tenía que llegar

Desde 1993 he estado encargado de la parte técnica de un pequeño ISP de acceso gratuito llamado Chester County InterLink (CCIL), en West Chester, Pennsylvania (fui uno de los fundadores de CCIL y escribí su original software BBS multiusuario, el cual puede verse entrando a telnet://locke.ccil.org. Actualmente soporta más de tres mil usuarios en 19 líneas). Este empleo me permitió tener acceso a la red las 24 horas del día a través de la línea de 56K de CCIL, ¡de hecho, prácticamente él me lo demandaba!

Para ese entonces ya me había habituado al correo electrónico. Por diversas razones fue difícil obtener SLIP para enlazar mi máquina en casa (snark.thyrsus.com) y CCIL. Cuando finalmente pude lograrlo, encontré que era particularmente molesto tener que entrar por telnet a locke cada rato para revisar mi correo. Lo que quería era que fuera reenviado a snark para que biff(1) me notificase cuando llegara.

Un simple redireccionamiento con sendmail no iba a funcionar debido a que snark no siempre está en línea y no tiene una dirección IP estática. Lo que necesitaba era un programa que saliera por mi conexión SLIP y trajera el correo hasta mi máquina. Yo sabía que tales programas ya existían, y que la mayoría usaba un protocolo simple llamado POP (Post Office Protocol, Protocolo de Oficina de Correos), de tal manera que me cercioré que el servidor POP3 estuviera en el sistema operativo BSD/OS de locke.

Necesitaba un cliente POP3; de tal manera que lo busqué en la red y encontré uno. En realidad hallé tres o cuatro. Usé pop-perl durante un tiempo, pero le faltaba una característica a todas luces evidente: la capacidad de identificar las direcciones de los correos recuperados para que las respuestas pudieran darse correctamente.

El problema era este: supongamos que un tal monty en locke me envía un correo. Si yo lo jalaba desde snark y luego intentaba responder, entonces mi programa de correos dirigía la respuesta a un monty inexistente en snark. La edición manual de las direcciones de respuesta para pegarles el ‘@ccil.org’, muy pronto se volvió algo muy molesto.

Era evidente que la computadora tenía que hacer esto por mí. (De hecho, de acuerdo con RFC1123, sección 5.2.18, sendmail debería de estarlo haciendo.) ¡Sin embargo, ninguno de los clientes POP lo hacía realmente! Esto nos lleva a la primera lección:

1. Todo buen trabajo de software comienza a partir de las necesidades personales del programador. (Todo buen trabajo empieza cuando uno tiene que rascarse su propia comezón).

Esto podría sonar muy obvio: el viejo proverbio dice que "la necesidad es la madre de todos los inventos". Empero, hay muchos programadores de software que gastan sus días, a cambio de un

salario, en programas que ni necesitan ni quieren. No ocurre lo mismo en el mundo Linux; lo que sirve para explicar por qué se da una calidad promedio de software tan alta en esa comunidad.

Por todo esto, ¿pensarán que me lancé inmediatamente a la vorágine de escribir, a partir de cero, el programa de un nuevo cliente POP3 que compitiese con los existentes? ¡Nunca en la vida! Revisé cuidadosamente las herramientas POP que tenía al alcance, preguntándome "¿cuál se aproxima más a lo que yo necesito?", porque

2. Los buenos programadores saben qué escribir. Los mejores, que reescribir (y reutilizar).

Aunque no presumo ser un extraordinario programador, he tratado siempre de imitar a uno de ellos. Una importante característica de los grandes programadores es la meticulosidad con la que construyen. Saben que les pondrán diez no por el esfuerzo, sino por los resultados; y que casi siempre será más fácil partir de una buena solución parcial que de cero.

Linus, por ejemplo, no intentó escribir Linux partiendo de cero. En vez de eso, comenzó por reutilizar el código y las ideas de Minix, un pequeño sistema operativo (OS) tipo UNIX hecho para máquinas 386. Eventualmente terminó desechando o reescribiendo todo el código del Minix, pero mientras ontó con él le sirvió como una importante plataforma de lanzamiento del proyecto en gestación que posteriormente se convertiría en Linux.

Con ese espíritu, comencé a buscar una herramienta POP que estuviese razonablemente escrita para ser usada como plataforma inicial para mi desarrollo.

La tradición del mundo UNIX de compartir las fuentes siempre se ha prestado a la reutilización del código (ésta es la razón por la que el proyecto GNU escogió a UNIX como su OS base, pese a las serias reservas que se tenían). El mundo Linux ha asumido esta tradición hasta llevarla muy cerca de su límite tecnológico; posee terabytes de código fuente que están generalmente disponibles.Así que es más probable que la búsqueda de algo bueno tenga mayores probabilidades de éxito en el mundo Linux que en ningún otro lado. Así sucedió en mi caso. Además de los que había encontrado antes, en mi segunda búsqueda conseguí un total de nueve candidatos: fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail y upop. El primero que elegí fue el ‘fetchpop’, un programa de Seung-Hong Oh. Le agregue mi código par que tuviera la capacidad de reescribir los encabezados y varias mejoras más, las cuales fueron incorporadas por el propio autor en la versión 1.9.

Sin embargo, unas semanas después me topé con el código fuente de ‘popclient’, escrito por Carl Harris, y descubrí que tenía un problema. Pese a que fetchpop poseía algunas ideas originales (tal como su modo demonio), sólo podía manejar POP3, y estaba escrito a la manera de un aficionado (Seung-Hong era un brillante programador, pero no tenía experiencia, y ambas características eran palpables). El código de Carl era mejor, bastante profesional y robusto, pero su programa carecía de varias de las características importantes del fetchpop que eran difíciles de implementar (incluyendo las que yo mismo había agregado).

¿Seguía o cambiaba? Cambiar significaba desechar el código que había añadido a cambio de una mejor base de desarrollo.

Un motivo práctico para cambiar fue la necesidad de contar con soporte de múltiples protocolos. POP3 es el protocolo de servidor de correos que más se utiliza, pero no es el único. Fetchpop y otros no manejaban POP2, RPOP ó APOP, y yo tenía ya la idea vaga de añadir IMAP (Protocolo de

Acceso a Mensajes por Internet, el protocolo de correos más poderoso y reciente) sólo por entretenimiento.

Pero había una razón más teórica para pensar que el cambio podía ser una buena idea, algo que aprendí mucho antes de Linux:

3. "Contemple desecharlo; de todos modos tendrá que hacerlo" (Fred Brooks, The Mythical Man-Month, Capítulo 11).

Diciéndolo de otro modo: no se entiende cabalmente un problema hasta que se implementa la primera solución. La siguiente vez quizáas uno ya sepa lo suficiente para solucionarlo. Así que si quieres resolverlo, disponte a empezar de nuevo al menos una vez.

Bien, me dije, los cambios a fetchpop fueron un primer intento, así que cambio.

Después de enviarle mi primera serie de mejoras a Carl Harris, el 25 de junio de 1996, me entere que él había perdido el interés por popclient desde hacía rato. El programa estaba un poco abandonado, polvoriento y con algunos bugs menores colgando. Como se le tenían que hacer varias correcciones, pronto acordamos que lo más lógico era que yo asumiera el control del proyecto.

Sin darme cuenta, el proyecto había alcanzado otras dimensiones. Ya no estaba intentando hacerle unos cuantos cambios menores a un cliente POP, sino que me había hecho responsable de uno; y las ideas que bullían en mi cabeza me conducirían probablemente a cambios mayores.

En una cultura del software que estimula el compartir el código fuente, ésta era la forma natural de que el proyecto evolucionara. Yo actuaba de acuerdo con lo siguiente:

4. Si tienes la actitud adecuada, encontrarás problemas interesantes.

Pero la actitud de Carl Harris fue aún más importante. Él entendió que

5. Cuando se pierde el interés en un programa, el último deber es heredarlo a un sucesor competente.

Sin siquiera discutirlo, Carl y yo sabíamos que el objetivo común era obtener la mejor solución. La única duda entre nosotros era si yo podía probar que el proyecto iba a quedar en buenas manos. Una vez que lo hice, él actuó de buena gana y con diligencia. Espero comportarme igual cuando llegue mi turno.

---

3. La importancia de contar con usuarios

Así fue como heredé popclient. Además, recibí su base de usuarios, lo cual fue tan o más importante. Tener usuarios es maravilloso. No sólo porque prueban que uno está satisfaciendo una necesidad, que ha hecho algo bien, sino porque, cultivados adecuadamente, pueden convertirse en magníficos asistentes.

Otro aspecto importante de la tradición UNIX, que Linux, de nuevo, lleva al límite, es que muchos de los usuarios son también hackers, y, al estar disponible el código fuente, se vuelven hackers muy efectivos. Esto puede resultar tremendamente útil para reducir el tiempo de depuración de los programas. Con un buen estímulo, los usuarios diagnosticarán problemas, sugerirán correcciones y ayudarán a mejor los programas mucho más rápido de lo que uno lo haría sin ayuda.

6. Tratar a los usuarios como colaboradores es la forma más apropiada de mejorar el código, y la más efectiva de depurarlo.

Suele ser fácil subestimar el poder de este efecto. De hecho, es posible que todos continuásemos desestimando la capacidad multiplicadora que adquiriría con el número de usuarios y en contra de la complejidad de los sistemas, hasta que así nos lo vino a demostrar Linus.

En realidad, considero que la genialidad de Linus no eradica en la construcción misma del kernel de Linux, sino en la invención del modelo de desarrollo de Linux. Cuando en una ocasión expresé esta opinión delante de él, sonrió y repitió quedito una frase que ha dicho muchas veces: "Básicamente soy una persona muy floja que le gusta obtener el crédito por lo que, realmente, hacen los demás". Flojo como una zorra. O, como diría Robert Heinlein, demasiado flojo para fallar.

En retrospectiva, un precedente de los métodos y el éxito que tiene Linux podría encontrarse en el desarrollo de las bibliotecas del Emacs GNU, así como los archivos del código de Lisp. En contraste con el estilo de construcción catedral del núcleo del Emacs escrito en C, y de muchas otras herramientas de la FSF, la evolución del código de Lisp fue bastante fluida y, en general, dirigida por los propios usuarios. Las ideas y los prototipos de los modos se reescribían tres o cuatro veces antes de alcanzar su forma estable final. Mientras que las frecuentes colaboraciones informales se hacían posibles gracias a la Internet, al estilo Linux.

Es más, uno de mis programas con mayor éxito, antes de fetchmail, fue probablemente el modo VC para Emacs, una colaboración tipo Linux, que realicé por correo electrónico conjuntamente con otras tres personas, de las cuales solamente he conocido a una (Richard Stallman) hasta la fecha. VC era una front-end para SCCS, RCS y posteriormente CVS, que ofrecía controles de tipo "al toque" para operaciones de control de versiones desde Emacs. Era el desarrollaba de un pequeño y, hasta cierto punto, rudimentario modo sccs.el que alguien había escrito. El desarrollo de VC tuvo éxito porque, a diferencia del Emacs mismo, el código de Emacs en Lisp podía pasar por el ciclo de publicar, probar y depurar, muy rápidamente.

(Uno de los efectos colaterales de la política de la FSF de atar legalmente el código a la GPL, fue que se volvió más difícil para la FSF usar el modo bazar, debido a su idea de que se debían de asignar derechos de autor por cada contribución individual de más de veinte líneas, a fin de inmunizar al código protegido por la GPL de cualquier problema legal surgido de ley de derechos de autor. Los usuarios de las licencias BSD y del MIT X Consortium no tienen este problema, debido a que no intentan reservarse derechos que cualquiera intente poner en duda.)

---

4. Libere rápido y a menudo

Las publicaciones rápidas y frecuentes del código constituyen una parte crítica del modelo Linux de desarrollo. La mayoría de los programadores, en los que me incluyo, creía antes que era una mala política involucrarse en proyectos más grandes triviales, debido a que las primeras versiones, casi por definición, salen plagadas de bugs, y a nadie le gusta agotar la paciencia de los usuarios.

Esta idea reafirmaba la preferencia de los programadores por el estilo catedral de desarrollo. Si el objetivo principal era que los usuarios vieran la menor cantidad de bugs, entonces sólo había que liberar una vez cada seis meses (o aún con menos frecuencia) y trabajar como burro en la depuración en el ínterin de las versiones que se saquen a la luz. El núcleo del Emacs escrito en C se

desarrolló de esta forma. No así la biblioteca de Lisp, ya que los repositorios de los archivos de Lisp, donde se podían conseguir versiones nuevas y en desarrollo del código, independientemente del ciclo de desarrollo del Emacs, estaban fuera del control de la FSF.

El más importante de estos archivos fue el elisp de la Universidad Estatal de Ohio, el cual se anticipó al espíritu y a muchas de las características de los grandes archivos actuales de Linux. Pero solamente algunos de nosotros reflexionamos realmente acerca de lo que estábamos haciendo, o de lo que la simple existencia del archivo sugería sobre los problemas implícitos en el modelo de desarrollo estilo catedral de la FSF. Yo realicé un intento serio, alrededor de 1992, de unir formalmente buena parte del código de Ohio con la biblioteca Lisp oficial del Emacs. Me metí en broncas políticas muy serias y no tuve éxito.

Pero un año después, a medida que Linux se agigantaba, quedo claro que estaba pasando algo distinto y mucho más sano. La política abierta de desarrollo de Linus era lo más opuesto a la construcción estilo catedral. Los repositorios de archivos en sunsite y tsx-11 mostraban una intensa actividad y muchas distribuciones de Linux circulaban. Y todo esto se manejaba con una frecuencia en la publicación de programas que no tenía precedentes. Linus estaba tratando a sus usuarios como colaboradores de la forma más efectiva posible:

7. Libere rápido y a menudo, y escuche a sus clientes.

La innovación de Linus no consistió tanto en esto (algo parecido había venido sucediendo en la tradición del mundo UNIX desde hacía tiempo), sino en llevarlo a un nivel de intensidad que estaba acorde con la complejidad de lo que estaba desarrollando. ¡En ese entonces no era raro que liberara una nueva versión del kernel más de una vez al día! Y, debido a que cultivó su base de desarrolladores asistentes y buscó colaboración en la Internet más intensamaente que ningún otro, funcionó.

¿Pero cómo fue que funcionó? ¿Era algo que yo podía emular, o se debía a la genialidad única de Linus?

No lo considero así. Está bien, Linus es un hacker endiabladamente astuto (¿cuántos de nosotros podrían diseñar un kernel de alta calidad?). Pero Linux en sí no representa ningún salto conceptual sorprendente hacia delante. Linus no es (al menos, no hasta ahora) un genio innovador del diseño como lo son Richard Stallman o James Gosling. En realidad, para mi Linus es un genio de la ingeniería; tiene un sexto sentido para evitar los callejones sin salida en el desarrollo y la depuración, y es tipo muy sagaz para encontrar el camino con el mínimo esfuerzo desde el punto A hasta el punto B. De hecho, todo el diseño de Linux transpira esta calidad, y refleja un Linus conservador que simplifica el enfoque en el diseño.

Por lo tanto, si las publicaciones frecuentes del código y la búsqueda de asistencia dentro de la Internet no son accidentes, sino partes integrales del ingenio de Linus para ver la ruta crítica del mínimo esfuerzo, ¿qué era lo que estaba maximizando? ¿Qué era lo que estaba exprimiendo de la maquinaria? Planteada de esta forma, las pregunta se responde por sí sola. Linus estaba manteniendo a sus usuarios-hackers-asistentes constantemente estimulados y recompensados por la perspectiva de tomar parte en la acción y satisfacer su ego, premiado con la exhibición y mejora constante, casi diaria, de su trabajo.

Linus apostaba claramente a maximizar el número de horas-hombre invertidas en la depuración y el desarrollo, a pesar del riesgo que corría de volver inestable el código y agotar a la base de usuarios, si un error serio resultaba insondable. Linus se portaba como si creyera en algo como esto:

8. Dada una base suficiente de desarrolladores asistentes y beta-testers, casi cualquier problema puede ser caracterizado rápidamente, y su solución ser obvia al menos para alguien.

O, dicho de manera menos formal, "con muchas miradas, todos los bugs saltarán a la vista". A esto lo he bautizado como la Ley de Linus. Mi formulación original rezaba que "todo problema deberá ser transparente para alguien". Linus descubrió que la personas que entendían y la que resolvían un problema no eran necesariamente las mismas, ni siquiera en la mayoría de los casos. Decía que "alguien encuentra el problema y otro lo resuelve". Pero el punto está en que ambas cosas suelen suceder con gran rapidez.

Aquí, pienso, subyace una diferencia esencial entre el estilo del bazar y el de la catedral. En el enfoque estilo catedral de la programación, los bugs y problemas de desarrollo son fenómenos truculentos, insidiosos y profundos. Generalmente toma meses de revisión exhaustiva para unos cuantos el alcanzar la seguridad de que han sido eliminados del todo. Por eso se dan los intervalos tan largos entre cada versión que se libera, y la inevitable desmoralización cuando estas versiones, largamente esperadas, no resultan perfectas.

En el enfoque de programación estilo bazar, por otro lado, se asume que los bugs son fenómenos relativamente evidentes o, por lo menos, que pueden volverse relativamente evidentes cuando se exhiben a miles de entusiastas desarrolladores asistentes que colaboran al parejo sobre cada una de las versiones. En consecuencia, se libera con frecuencia para poder obtener una mayor cantidad de correcciones, logrando como efecto colateral benéfico el perder menos cuando un eventual obstáculo se atraviesa.

Y eso es todo. Con eso basta. Si la Ley de Linus fuera falsa, entonces cualquier sistema suficientemente complejo como el kernel de Linux, que está siendo manipulado por tantos, debería haberse colapsado en un punto bajo elpeso de ciertas interacciones imprevistas y bugs "muy profundos" inadvertidos. Pero si es cierta, bastaría para explicar la relativa ausencia de bugs el código de Linux.

Después de todo, esto no debí parecernos tan sorpresivo. Hace algunos años los sociólogos descubrieron que la opinión promedio de un numero grande de observadores igualmente expertos (o igualmente ignorantes) es más confiable de predecir que la de uno de los observadores seleccionado al azar. A esto se le conoce como el efecto Delphi. Al parecer, lo que Linus ha demostrado es que esto también es valedero en el ámbito de la depuración de un sistema operativo: que el efecto Delphi puede abatir la complejidad implícita en el desarrollo, incluso al nivel de la involucrada en el desarrollo del núcleo de un OS.

Estoy en deuda con Jeff Dutky dutky@wam.umd.edu, quien me sugirió que la Ley de Linus puede replantearse diciendo que "la depuración puede hacerse en paralelo". Jeff señala que a pesar de que la depuración requiere que los participantes se comuniquen con un programador que coordina el trabajo, no demanda ninguna coordinación significativa entre ellos. Por lo tanto, no cae víctima de la asombrosa complejidad cuadrática y los costos de maniobra que ocasionan que la incorporación de desarrolladores resulte problemática.

En la práctica, la pérdida teórica de eficiencia debido a la duplicación del trabajo por parte de los programadores casi nunca es un tema que revista importancia en el mundo Linux. Un efecto de la "política de liberar rápido y a menudo" es que esta clase de duplicidades se minimizan al propagarse las correcciones rápidamente.

Brooks hizo una observación relacionada con la de Jeff: "El costo total del mantenimiento de un programa muy usado es típicamente alrededor del 40 por ciento o más del costo del desarrollo. Sorpresivamente, este costo está fuertemente influenciado por el número de usuarios. Más usuarios detectan una mayor cantidad de bugs" (El subrayado es mío). Una mayor cantidad de usuarios detecta más bugs debido a que tienen diferentes maneras de evaluar el programa. Este efecto se incrementa cuando los usuarios son desarrolladores asistentes. Cada uno enfoca la tarea de la caracterización de los bugs con un bagaje conceptual e instrumentos analíticos distintos, desde un ángulo diferente. El efecto Delphi parece funcionar precisamente debido a estas diferencias. En el contexto específico de la depuración, dichas diferencias también tienden a reducir la duplicación del trabajo.

Por lo tanto, el agregar más beta-testers podría no contribuir a reducir la complejidad del más profundo de los errores actuales, desde el punto de vista del desarrollador, pero aumenta la probabilidad de que la caja de herramientas de alguno de ellos se equipare al problema, de tal suerte que esa persona vea claramente el error.

Linus también dobla sus apuestas. En el caso de que realmente existan bugs serios, las versiones del kernel de Linux son enumeradas de tal manera que los usuarios potenciales puedan escoger la última versión considerada como "estable" o ponerse al filo de la navaja y arriesgarse a los bugs tal de aprovechar las nuevas características. Esta táctica no ha sido formalmente imitada por la mayoría de los hackers de Linux, pero quizá deberían hacerlo. El hecho de contar con ambas opciones, lo vuelve aún más atractivo.

---

5. ¿Cuándo una rosa no es rosa?

Después de estudiar la forma en que actuó Linus y haber formulado una teoría del por qué tuvo éxito, tomé la decisión consciente de probarla en mi nuevo proyecto (el cual, debo admitirlo, es mucho menos complejo y ambicioso).

Lo primero que hice fue reorganizar y simplificar popclient. El trabajo de Carl Harris era muy bueno, pero exhibía una complejidad innecesaria, típica de muchos de los programadores en C. Él trataba el código como la parte central y las estructuras de datos como un apoyo para éste. Como resultado, el código resultó muy elegante, pero el diseño de las estructuras de datos salió ad hoc y feo (por lo menos con respecto a los estándares exigentes de este viejo hacker de Lisp).

Sin embargo, tenía otro motivo para reescribir, además de mejorar el diseño de la estructura de datos y el código: El proyecto debía evolucionar en algo que yo entendiera cabalmente. No es nada divertido ser el responsable de corregir los errores en un programa que no se entiende. Por lo tanto, durante el primer mes, o algo así, simplemente fui siguiendo los pormenores del diseño básico de Carl. El primer cambio serio que realicé fue agregar el soporte de IMAP. Lo hice reorganizando los administradores de protocolos en un administrador genérico con tres tablas de métodos (para POP2, POP3 e IMAP). Éste y algunos cambios anteriores muestran un principio general que es bueno que

los programadores tengan en mente, especialmente los que programan en lenguajes tipo C y no hacen manejo de datos dinámicamente:

9. Las estructuras de datos inteligentes y el código burdo funcionan mucho mejor que en el caso inverso.

De nuevo, Fred Brooks, Capítulo 11: "Muéstreme su código y esconda sus estructuras de datos, y continuaré intrigado. Muéstreme sus estructuras de datos y generalmente no necesitaré ver su código; resultará evidente".

En realidad, él hablaba de "diagramas de flujo" y "tablas". Pero, con treinta años de cambios terminológicos y culturales, resulta prácticamente la misma idea.

En este momento (a principios de septiembre de 1996, aproximadamente seis semanas después de haber comenzado) empecé a pensar que un cambio de nombre podría ser apropiado. Después de todo, ya no se trataba de un simple cliente POP. Pero todavía vacilé, debido a que no había nada nuevo y genuinamente mío en el diseño. Mi versión del popclient tenía aún que desarrollar una identidad propia.

Esto cambio radicalmente cuando fetchmail aprendió a remitir el correo recibido al puerto SMTP. Volveré a este punto en un momento. Primero quiero decir lo siguiente: yo afirmé anteriormente que decidí utilizar este proyecto para probar mi teoría sobre la corrección del estilo Linus Torvalds.

¿Cómo lo hice? (podrían ustedes preguntar muy bien). Fue de la siguiente manera:

1. 1. Liberaba rápido y a menudo (casi nunca dejé de hacerlo en menos de diez días; durante los períodos de desarrollo intenso, una vez diaria).
2. 2. Ampliaba mi lista de analistas de versiones beta, incorporando a todo el que me contactara para saber sobre fetchmail.
3. 3. Efectuaba anuncios espectaculares a esta lista cada vez que liberaba una nueva versión, estimulando a la gente a participar.
4. 4. Y escuchaba a mis analistas asistentes, consultándolos decisiones referentes al diseño y tomándolos en cuenta cuando me mandaban sus mejoras y la consecuente retroalimentación.

La recompensa por estas simples medidas fue inmediata. Desde el principio del proyecto obtuve reportes de errores de calidad, frecuentemente con buenas soluciones anexas, que envidiarían la mayoría de los desarrolladores. Obtuve crítica constructiva, mensajes de admiradores e inteligentes sugerencias. Lo que lleva a la siguiente lección: ***10. Si usted trata a sus analistas (beta-testers) como si fueran su recurso más valioso, ellos le responderán convirtiéndose en su recurso más valioso.***

Una medida interesante del éxito de fetchmail fue el tamaño de la lista de analistas beta del proyecto, los amigos de fetchmail. Cuando escribí esto, tenía 249 miembros, y se sumaban entre dos y tres semanalmente.

Revisándola hoy, finales de mayo de 1997, la lista ha comenzando a perder miembros debido a una razón sumamente interesante. ¡Varias personas me han pedido que los dé de baja debido a que el fetchmail les está funcionando tan bien que ya no necesitan ver todo el tráfico de de la lista! A lo mejor esto es parte del ciclo vital normal de un proyecto maduro realizado por el método de construcción estilo bazar.

---

6. Popclient se convierte en fetchmail

El momento crucial para el proyecto fue cuando Harry Hochheiser me mandó su código fuente para incorporar la remisión del correo recibido a la máquina cliente a través del puerto SMTP. Comprendí casi inmediatamente que una implementación adecuada de esta característica iba a dejar a todos los demás métodos a un paso de ser obsoletos.

Durante muchas semanas había estado perfeccionando fetchmail, agregándole características, a pesar de que sentía que el diseño de la interfaz era útil pero algo burdo, poco elegante y con demasiadas opciones insignificantes colgando fuera de lugar. La facilidad de vaciar el correo recibido a un archivo-buzón de correos o la salida estándar me incomodaba de cierta manera, pero no alcanzaba a comprender por qué.

Lo que advertí cuando me puse a pensar sobre la expedición del correo por el SMTP fue que el popclient estaba intentando hacer demasiadas cosas juntas. Había sido diseñado para funcionar al mismo tiempo como un agente de transporte (MTA) y un agente de entrega (MDA). Con la remisión del correo por el SMTP podría abandonar la función de MDA y centrarme solamente en la de MTA, mandando el correo a otros programas para su entrega local, justo como lo hace sendmail.

¿Por qué sufrir con toda la complejidad que encierra ya sea configurar el agente de entrega o realizar un bloqueo y luego un añadido al final del archivo-buzón de correos, cuando el puerto 25 está casi garantizado casi en toda plataforma con soporte TCP/IP? Especialmente cuando esto significa que el correo obtenido de esta manera tiene garantizado verse como un correo que ha sido transferido de manera normal, por el SMTP, que es lo que realmente queremos.

De aquí se extraen varias lecciones. Primero, la idea de enviar por el puerto SMTP fue la mayor recompensa individual que obtuve al tratar de emular conscientemente los métodos de Linus. Un usuario me proporcionó una fabulosa idea, y lo único que restaba era comprender sus implicaciones.

11. Lo más grande, después de tener buenas ideas, es reconocer las buenas ideas de sus usuarios. Esto último es a veces lo mejor. Lo que resulta muy interesante es que usted rápidamente encontrará que cuando esta absolutamente convencido y seguro de lo que le debe a los demás, entonces el mundo lo tratará como si usted hubiera realizado cada parte de la invención por si mismo, y esto le hará apreciar con modestia su ingenio natural. ¡Todos podemos ver lo bien que funcionó esto para el propio Linus!

(Cuando leía este documento en la Conferencia de Perl de agosto de 1997, Larry Wall estaba en la fila del frente. Cuando llegué a lo que acabo de decir, Larry dijo con voz alta: "¡Anda, di eso, díselos, hermano!". Todos los presentes rieron porque sabían que eso también le había funcionado muy bien al inventor de Perl.)

Y a unas cuantas semanas de haber echado a andar el proyecto con el mismo espíritu, comencé a recibir adulaciones similares, no sólo de parte de mis usuarios, sino de otra gente que se había enterado por terceras personas.

He puesto a buen recaudo parte de ese correo. Lo volveréa a leer en alguna ocasión, si es que me llego a preguntar si mi vida ha valido la pena :-). Pero hay otras dos lecciones más fundamentales, que no tienen que ver con las políticas, que son generales para todos los tipos de diseño:

12. Frecuentemente, las soluciones más innovadoras y espectaculares provienen de comprender que la concepción del problema era errónea.

Había estado intentando resolver el problema equivocado al continuar desarrollando el popclient como un agente de entrega y de transporte combinados, con toda clase de modos medio raros de entrega local. El diseño de fetchmail requería ser repensado de arriba abajo como un agente de transporte puro, como eslabón, si se habla de SMTP, de la ruta normal que sigue el correo en Internet.

Cuando usted se topa con un muro durante el desarrollo —cuando la encuentra difícil como para pensar mas allá de la corrección que sigue— es, a menudo, la hora de preguntarse no si usted realmente tiene la respuesta correcta, sino si se está planteando la pregunta correcta. Quizás el problema requiere ser replanteado.

Bien, yo ya había replanteado mi problema. Evidentemente, lo que tenía que hacer ahora era (1) programar el soporte de envío por SMTP en el controlador genérico, (2) hacerlo el modo por omisión, y (3) eliminar eventualmente todas las demás modalidades de entrega, especialmente las de envío a un archivo- buzón y la de vaciado a la salida estándar.

Estuve, durante algún tiempo, titubeando en dar el paso 3; temiendo trastornar a los viejos usuarios de poclient, quienes dependían de estos mecanismos alternativos de entrega. En teoría, ellos podían cambiar inmediatamente a archivos .forward, o sus equivalentes en otro esquema que no fuera sendmail, para obtener los mismos resultados. Pero, en la práctica, la transición podría complicarse demasiado.

Cuando por fin lo hice, empero, los beneficios fueron inmensos. Las partes más intrincadas del código del controlador desaparecieron. La configuración se volvió radicalmente más simple: al no tratar con el MDA del sistema y con el archivo-buzón del usuario, ya no había que preocuparse de que el sistema operativo soportara bloqueo de archivos. Asimismo, el único riesgo de extraviar correo también se había desvanecido. Antes, si usted especificaba el envío a un archivo-buzón y el disco estaba lleno, entonces el correo se perdía irremediablemente. Esto no pasa con el envío vía SMTP debido a que el SMTP del receptor no devolverá un OK mientras el mensaje no haya sido entregado con éxito, o al menos haya sido mandado a la cola para su entrega ulterior.

Además, el desempeño mejoró mucho (aunque uno no lo notará en la primera corrida). Otro beneficio nada despreciable fue la simplificación de la página del manual.

Más adelante hubo que agregar la entrega a un agente local especificado por el usuario con el fin de manejar algunas situaciones oscuras involucradas con la asignación dinámica de direcciones en SLIP. Sin embargo, encontré una forma mucho más simple de hacerlo.

¿Cuál era la moraleja? No hay que vacilar en desechar alguna característica superflua si puede hacerlo sin pérdida de efectividad. Antôine de Saint- Exupery (un aviador y diseñador de aviones, cuando no se dedicaba a escribir libros clásicos para niños) afirmó que

13. "La perfección (en diseño) se alcanza no cuando ya no hay nada que agregar, sino cuando ya no hay algo que quitar".

Cuando el código va mejorando y se va simplificando, es cuando sabe que está en lo correcto. Así, en este proceso, el diseño de fetchmail adquirió una identidad propia, diferente de su ancestro, el popclient.

Había llegado la hora de cambiar de nombre. El nuevo diseño parecía más un doble del Sendmail que el viejo popclient; ambos eran MTAs, agentes de transporte, pero mientras que el Sendmail

empuja y luego entreg, el nuevo popclient jala y después entrega. Así que, después de dos arduos meses, lo bautice de nuevo con el nombre de fetchmail.

---

7. El crecimiento de fetchmail

Allí me encontraba con un bonito e innovador diseño, un programa que sabía funcionaba bien porque lo utilizaba diariamente, y me enteraba por la lista beta, que era muy activa. Esta gradualmente me hizo ver que ya no estaba involucrado en un hackeado personal trivial, que podía resultar útil para unas cuantas personas más. Tenía en mis manos un programa que cualquier hacker con una caja UNIX y una conexión SLIP/PPP realmente necesita.

Con el método de expedición por SMTP se puso adelante de la competencia, lo suficiente como para poder convertirse en un "matón profesional", uno de esos programas clásicos que ocupa tan bien su lugar que las otras alternativas no sólo son descartadas, sino olvidadas. Pienso que uno realmente no podría imaginar o planear un resultado como éste. Usted tiene que meterse a manejar conceptos de diseño tan poderosos que posteriormente los resultados parezcan inevitables, naturales, o incluso predestinados. La única manera de hacerse de estas ideas es jugar con un montón de ideas; o tener una visión de la ingeniería suficiente para poder llevar las buenas ideas de otras personas más allá de lo que sus propios autores originales pensaban que podían llegar.

Andrew Tanenbaum tuvo una buena idea original, con la construcción de un UNIX nativo simple para 386, que sirviera como herramienta de enseñanza. Linus Torvalds llevó el concepto de Minix más allá de lo que Andrew imagino que pudiera llegar, y se transformó en algo maravilloso. De la misma manera (aunque en una escala menor), tomé algunas ideas de Carl Harris y Harry Hochheiser y las impulsé fuertemente. Ninguno de nosotros era original en el sentido romántico de la idea que la gente tiene de un genio.

Pero, la mayor parte del desarrollo de la ciencia, la ingeniería y el software no se debe a un genio original, sino a la mitología del hacker por el contrario.

Los resultados fueron siempre un tanto complicados: de hecho, ¡justo el tipo de reto para el que vive un hacker! Y esto implicaba que tenía que fijar aún más alto mis propios estándares. Para hacer que el fetchmail fuese tan bueno como ahora veía que podía ser, tenía que escribir no sólo para satisfacer mis propias necesidades, sino también incluir y dar el soporte a otros que estuvieran fuera de mi órbita. Y esto lo tenía que hacer manteniendo el programa sencillo y robusto.

La primera característica más importante y contundente que escribí después de hacer eso fue el soporte para recabado múltiple, esto es, la capacidad de recoger el correo de los buzones que habían acumulado todo el correo de un grupo de usuarios, y luego trasladar cada mensaje al recipiente individual del respectivo destinatario.

Decidí agregarle el soporte de recabado múltiple debido en parte a que algunos usuarios lo reclamaban, pero sobre todo porque evidenciaría los bugs de un código de recabado individual, al forzarme a abordar el direccionamiento con generalidad. Tal como ocurrió. Poner el RFC822 a que funcionara correctamente me tomó bastante tiempo, no sólo porque cada uno de las partes que lo componen son difíciles, sino porque involucraban un montón de detalles confusos e interdependientes entre sí.

Así, pues, el direccionamiento del recabado múltiple se volvió una excelente decisión de diseño. De esta forma supe que:

14. Toda herramienta es útil empleándose de la forma prevista, pero una "gran" herramienta es la que se presta a ser utilizada de la manera menos esperada.

El uso inesperado del recabado múltiple del fetchmail fue el trabajar listas de correo con la lista guardada, y realizar la expansión del alias en el lado del cliente de la conexión SLIP/PPP. Esto significa que alguien que cuenta con una computadora y una cuenta de ISP puede manejar una lista de correos sin que tenga que continuar entrando a los archivos del alias del ISP. Otro cambio importante reclamado por mis auxiliares beta era el soporte para la operación MIME de 8 bits. Esto se podía obtener fácilmente, ya que había tenido cuidado de mantener el código de 8 bits limpio. No es que yo me hubiera anticipado a la exigencia de esta característica, sino que obedecía a otra regla:

15. Cuándo se escribe software para una puerta de enlace de cualquier tipo, hay que tomar la precaución de alterar el flujo de datos lo menos posible, y ¡*nunca* eliminar información a menos que los receptores obliguen a hacerlo!

Si no hubiera obedecido esta regla, entonces el soporte MIME de 8 bits habría resultado difícil y lleno de bugs. Así las cosas, todo lo que tuve que hacer fue leer el RFC 1652 y agregar algo de lógica trivial en la generación de encabezados.

Algunos usuarios europeos me presionaron para que introdujera una opción que limitase el número de mensajes acarreados por sesión (de manera tal que pudieran controlar los costos de sus redes telefónicas caras). Me opuse a dicho cambio durante mucho tiempo, y aun no estoy totalmente conforme con él. Pero si usted escribe para el mundo, debe escuchar a sus clientes: esto no debe cambiar en nada tan sólo porque no le están dando dinero.

---

8 Algunas lecciones mas extraídas de fetchmail

Antes de volver a los temas generales de ingeniería de software, hay que ponderar otras dos lecciones específicas sacadas de la experiencia de fetchmail.

La sintaxis de los archivos rc incluyen palabras clave opcionales "de ruido" que son ignoradas totalmente por el analizador de sintaxis. La sintaxis tipo inglés que estas permiten es considerablemente más legible que la secuencia de pares palabra clave-valor tradicionales que usted obtiene cuando quita esas palabras clave opcionales.

Estas comenzaron como un experimento de madrugada, cuando noté que muchas de las declaraciones de los archivos rc se asemejaban un poco a un minilenguaje imperativo. (Esta también fue la razón por la cual cambié la palabra clave original del popclient de "servidor" a "poll"). Me parecía en ese entonces que el convertir ese minilenguaje imperativo más tipo inglés lo podía hacer más fácil de usar. Ahora, a pesar de que soy un partidario convencido de la escuela de diseño "hágalo un lenguaje", ejemplificada en Emacs, HTML y muchas bases de datos, no soy normalmente un fanático de la sintaxis estilo inglés.

Los programadores han tendido a favorecer tradicionalmente la sintaxis de control, debido a que es muy precisa, compacta y no tienen redundancia alguna. Esto es una herencia cultural de la época en que los recursos de cómputo eran muy caros, por lo que la etapa de análisis tenía que ser la más

sencilla y económica posible. El inglés, con un 50% de redundancia, parecía ser un modelo muy inapropiado en ese entonces.

Este no es la razón por la cual yo dudo de la sintaxis tipo inglés; y sólo lo menciono aquí para demolerlo. Con los ciclos baratos, la fluidez no debe ser un fin por sí misma. Ahora es más importante para un lenguaje el ser conveniente para los humanos que ser económico en términos de recursos computacionales.

Sin embargo, hay razones suficientes para andar con cuidado. Una es el costo de la complejidad de la etapa de análisis: nadie quiere incrementarlo a un punto tal que se vuelva una fuente importante de bugs y confusión para el usuario. Otra radica en que al hacer una sintaxis del lenguaje tipo inglés se exige frecuentemente que se deforme considerablemente el "inglés" que habla, por lo que la semejanza superficial con un lenguaje natural es tan confusa como podría haberlo sido la sintaxis tradicional. (Usted puede ver mucho de esto en los 4GLs y en los lenguajes de búsqueda en bancos de datos comerciales).

La sintaxis de control de fetchmail parece esquivar estos problemas debido a que el dominio de su lenguaje es extremadamente restringido. Está muy lejos de ser un lenguaje de amplio uso; las cosas que dice no son muy complicadas, por lo que hay pocas posibilidades de una confusión, al moverse de un reducido subconjunto del inglés y el lenguaje de control real. Creo que se puede extraer una lección más general de esto:

16. Cuando su lenguaje está lejos de un Turing completo, entonces el azúcar sintáctico puede ser su amigo.

Otra lección trata de la seguridad por obscuridad. Algunos usuarios de fetchmail me solicitaron cambiar el software para poder guardar las claves de acceso encriptadas en su archivo rc, de manera tal que los crackers no pudieran verlos por pura casualidad.

No lo hice debido a que esto prácticamente no proporcionaría ninguna protección adicional. Cualquiera que adquiera los permisos necesarios para leer el archivo rc respectivo sería de todos modos capaz de correr el fetchmail; y si por su password fuera, podría sacar el decodificador necesario del mismo código del fetchmail para obtenerlo.

Todo lo que la encriptación de passwords en el archivo .fetchmailrc podría haber conseguido era una falsa sensación de seguridad para la gente que no está muy metida en este medio. La regla general es la siguiente:

17. Un sistema de seguridad es tan seguro como secreto. Cuídese de los secretos a medias.

---

9. Condiciones necesarias para el estilo del bazar

Los primeros que leyeron este documento, así como sus primeras versiones inacabadas que se hicieron públicas, preguntaban constantemente sobre los requisitos necesarios para un desarrollo exitoso dentro del modelo del bazar, incluyendo tanto la calificación del líder del proyecto como la del estado del código cuando uno va a hacerlo público y a comenzar a construir una comunidad de co-desarrolladores.

Está claro que uno no puede partir de cero en el estilo bazar. Con él, uno puede probar, buscar errores, poner a punto y mejorar algo, pero sería muy difícil "originar" un proyecto en un modo

semejante al bazar. Linus no lo intentó de esta manera. Yo tampoco lo hice así. Nuestra naciente comunidad de desarrolladores necesita algo que ya corra para jugar.

Cuando uno comienza la construcción del edificio comunal, lo que debe ser capaz de hacer es presentar una "promesa plausible". El programa no necesita ser particularmente bueno. Puede ser burdo, tener muchos bugs, estar incompleto y pobremente documentado. Pero en lo que no se puede fallar es en convencer a los co-desarrolladores potenciales de que el programa puede evolucionar hacia algo elegante en el futuro.

Linux y fetchmail se hicieron públicos con diseños básicos fuertes y atractivos. Mucha gente piensa que el modelo del bazar tal como lo he presentado, ha considerado correctamente esto como crítico, y luego ha saltado de aquí a la conclusión de que es indispensable que el líder del proyecto tenga un mayor nivel de intuición para el diseño y mucha capacidad.

Sin embargo, Linus obtuvo su diseño a partir de UNIX. Yo inicialmente conseguí el mío del antiguo popmail (a pesar de que cambiaría mucho posteriormente, mucho más, guardando las proporciones, de lo que lo ha hecho Linux). Entonces, ¿tiene que poseer realmente un talento extraordinario el líder-coordinador en el modelo del bazar, o la puede ir pasando con tan sólo coordinar el talento de otros para el diseño?

Creo que no es crítico que el coordinador sea capaz de originar diseños de calidad excepcional, pero lo que sí es absolutamente esencial es que él (o ella) sea capaz de "reconocer las buenas ideas sobre diseño de los demás".

Tanto el proyecto de Linux como el de fetchmail dan evidencias de esto. A pesar de que Linus no es un diseñador original espectacular (como lo discutimos anteriormente), ha mostrado tener una poderosa habilidad para reconocer un buen diseño e integrarlo al kernel de Linux. Ya he descrito cómo la idea de diseño de mayor envergadura para el fetchmail (reenvío por SMTP) provino de otro.

Los primeros lectores de este artículo me halagaron al sugerir que soy propenso a subestimar la originalidad en el diseño en los proyectos del bazar, debido a que la tengo en buena medida, y por lo tanto, la tomo por sentada. Puede ser verdad en parte; el diseño es ciertamente mi fuerte (comparado con la programación o la depuración). Pero el problema de ser listo y original en el diseño de software se tiende a convertir en hábito: uno hace las cosas como por reflejo, de manera tal que parezcan elegantes y complicadas, cuando debería mantenerlas simples y robustas. Ya he sufrido tropiezos en proyectos debido a esta equivocación, pero me las ingenié para no sucediera lo mismo con fetchmail. Así, pues, considero que el proyecto del fetchmail tuvo éxito en parte debido a que contuve mi propensión a ser astuto; este es un argumento que va (por lo menos) contra la originalidad en el diseño como algo esencial para que los proyectos del bazar sean exitosos. Consideremos de nuevo Linux. Supóngase que Linus Torvalds hubiera estado tratando de desechar innovaciones fundamentales en el diseño del sistema operativo durante la etapa de desarrollo; ¿podría acaso ser tan estable y exitoso como el kernel que tenemos hoy en realidad?

Por supuesto, se necesita un cierto nivel mínimo de habilidad para el diseño y la escritura de programas, pero es de esperar que cualquiera que quiera seriamente lanzar un esfuerzo al estilo del bazar ya esté por encima de este nivel. El mercado interno de la comunidad del software libre, por reputación, ejerce una presión sutil sobre la gente para que no inicie esfuerzos de desarrollo que no sea capaz de mantener. Hasta ahora, esto parece estar funcionando bastante bien.

Existe otro tipo de habilidad que no esta asociada normalmente con el desarrollo del software, la cual yo considero que es igual de importante para los proyectos del bazar, y a veces hasta más, como el ingenio en el diseño. Un coordinador o líder de proyecto estilo bazar debe tener don de gentes y una buena capacidad de comunicación.

Esto podría parecer obvio. Para poder construir una comunidad de desarrollo se necesita atraer gente, interesarla en lo que se está haciendo y mantenerla a gusto con el trabajo que se está desarrollando. El entusiasmo técnico constituye una buena parte para poder lograr esto, pero está muy lejos de ser definitivo. Además, es importante la personalidad que uno proyecta.

No es una coincidencia que Linus sea un tipo que hace que la gente lo aprecie y desee ayudarle. Tampoco es una coincidencia que yo sea un extrovertido incansable que disfruta de trabajar con una muchedumbre, y tenga un poco de porte e instintos de cómico improvisado. Para hacer que el modelo bazar funcione, ayuda mucho tener al menos un poco de capacidad para las relaciones sociales.

---

10. El contexto social del software libre

Bien se ha dicho: las mejores hackeadas comienzan como soluciones personales a los problemas cotidianos del autor, y se se vuelven populares debido a que el problema común para un buen grupo de usuarios. Esto nos hace regresar al tema de la regla 1, que quizá puede replantearse de una manera más útil: ***18. Para resolver un problema interesante, comience por encontrar un problema que le resulte interesante.***

Así ocurrió con Carl Harris y el antiguo popclient, y así sucede conmigo y fetchmail. Esto, sin embargo, se ha entendido desde hace mucho. El punto interesante, el punto que las historias de Linux y fetchmail nos piden enfocar, está en la siguiente etapa, en la de la evolución del software en presencia de una amplia y activa comunidad de usuarios y co-desarrolladores.

En The Mythical Man-Month, Fred Brooks observó que el tiempo del programador no es fungible; que el agregar desarrolladores a un proyecto maduro de software lo vuelve tardío. Expuso que la complejidad y los costos de comunicación de un proyecto aumentan como el cuadrado del número de desarrolladores, mientras que el trabajo crece sólo linealmente. A este planteamiento se le conoce como la Ley de Brooks, y es generalmente aceptado como algo cierto. Pero si la Ley de Brooks fuese general, entonces Linux sería imposible.

Unos años después, el clásico de Gerald Weinberg La Psicología de la Programación de Computadoras plantea, visto en retrospectiva, una corrección esencial a Brooks. En su discusión de la "programación sin ego", Weinberg señala que en los lugares donde los desarrolladores no tienen propiedad sobre su código, y estimulan a otras personas a buscar bugs y posibles mejoras, son los lugares donde el avance es dramáticamente más rápido que en cualquier otro lado.

La terminología empleada por Weinberg ha evitado quizá que su análisis gane la aceptación que merece: uno tiene que sonreír al oír que los hackers de Internet no tienen ego. Creo, no obstante, que su argumentación parece más valida ahora que nunca.

La historia de UNIX debió habernos preparado para lo que hemos aprendido de Linux (y lo que he verificado experimentalmente en una escala más reducida al copiar deliberadamente los métodos de

Linus). Esto es, mientras que la creación de programas sigue siendo esencialmente una actividad solitaria, los desarrollos realmente grandes surgen de la atención y la capacidad de pensamiento de comunidades enteras. El desarrollador que usa solamente su cerebro sobre un proyecto cerrado está quedando detrás del que sabe como crear en un contexto abierto y evolutivo en el que la búsqueda de bugs mejoras son realizadas por cientos de personas.

Pero el mundo tradicional de UNIX no pudo llevar este enfoque hasta sus últimas consecuencias debido a varios factores. Uno era las limitaciones legales producidas por varias licencias, secretos e intereses comerciales. Otra (en retrospectiva) era que la Internet no estaba todavía madura para lograrlo.

Antes de que Internet fuera tan accesible, había comunidades geográficamente compactas en las cuales la cultura estimulaba la "programación sin ego" de Weinberg, y el desarrollador podía atraer fácilmente a muchos desarrolladores y usuarios capacitados. El Bell Labs, el MIT AI Lab, la Universidad de California en Berkeley son lugares donde se originaron innovaciones que son legendarias y aún poderosas.

Linux fue el primer proyecto de un esfuerzo consciente y exitoso de usar el mundo entero como un nido de talento. No creo que sea coincidencia que el período de gestación de Linux haya coincidido con el nacimiento de la World Wide Web, y que Linux haya dejado su infancia durante el mismo período, en 1993-1994, en que se vio el despegue de la industria ISP y la explosión del interés masivo por la Internet. Linus fue el primero que aprendió a jugar con las nuevas reglas que esa Internet penetrante hace posibles.

A pesar de que la Internet barata es una condición necesaria para que evolucionara el modelo de Linux, no creo que sea en sí misma una condición suficiente. Otros factores vitales fueron el desarrollo de un estilo de liderazgo y el arraigo de hábitos cooperativos, que permiten a los programadores atraer más co-desarrolladores y obtener el máximo provecho del medio.

Pero, ¿qué es el estilo de liderazgo y qué estos hábitos? No pueden estar basados en relaciones de poder, y aunque lo fueran, el liderazgo por coerción no produciría los resultados que estamos viendo. Weinberg cita un pasaje de la autobiografía del anarquista ruso del siglo XIX Kropotkin Memorias de un revolucionario, que está muy acorde con este tema:

"Habiendo sido criado en una familia que tenía siervos, me incorporé a la vida activa, como todos los jóvenes de mi época, con una gran confianza en la necesidad de mandar, ordenar, regañar, castigar y cosas semejantes. Pero cuando, en una etapa temprana, tuve que manejar empresas serias y tratar con hombres libres, y cuando cada error podría acarrear serias consecuencias, yo comencé a apreciar la diferencia entre actuar con base en el principio de orden y disciplina y actuar con base en el principio del entendimiento. El primero funciona admirablemente en un desfile militar, pero no sirve cuando está involucrada la vida real y el objetivo sólo puede lograrse mediante el esfuerzo serio de muchas voluntades convergentes".

El "esfuerzo serio de muchas voluntades convergentes" es precisamente lo que todo proyecto estilo Linux requiere; mientras que el "principio de orden y disciplina" es efectivamente imposible de aplicar a los voluntarios del paraíso anarquista que llamamos Internet. Para poder trabajar y competir de manera efectiva, los hackers que quieran encabezar proyectos de colaboración deben aprender a reclutar y entusiasmar a las comunidades de interés de un modo vagamente sugerido por el "principio de entendimiento" de Kropotkin. Deben aprender a usar la Ley de Linus.

Anteriormente me referí al efecto Delphi como una posible explicación de la Ley de Linus. Pero existen analogías más fuertes con sistemas adaptativos en biología y economía que se sugieren irresistiblemente. El mundo de Linux se comporta en muchos aspectos como el libre mercado o un sistema ecológico, donde un grupo de agentes individualistas buscan maximizar la utilidad en la que los procesos generan un orden espontáneo autocorrectivo más desarrollado y eficiente que lo que podría lograr cualquier tipo de planeación centralizada. Aquí, entonces, es el lugar para ver el "principio del entendimiento".

La "función utilidad" que los hackers de Linux están maximizando no es económica en el sentido clásico, sino algo intangible como la satisfacción de su ego y su reputación entre otros hackers. (Uno podría hablar de su "motivación altruista", pero ignoraríamos el hecho de que el altruismo en sí mismo es una forma de satisfacción del ego para el altruista). Los grupos voluntarios que funcionan de esta manera no son escasos realmente; uno en el que he participado es el de los aficionados a la ciencia ficción, que a diferencia del mundo de los hackers, reconoce explícitamente el "egoboo" (el realce de la reputación de uno entre los demás) como la motivación básica que está detrás de la actividad de los voluntarios.

Linus, al ponerse exitosamente como vigía de un proyecto en el que el desarrollo es realizado por otros, y al alimentar el interés en él hasta que se hizo autosustentable, ha mostrado el largo alcance del "principio de entendimiento mutuo" de Kropotkin. Este enfoque cuasieconómico del mundo de Linux nos permite ver cual es la función de tal entendimiento.

Podemos ver al método de Linus como la forma de crear un mercado eficiente en el "egoboo", que liga, lo más firme posible, el egoísmo de los hackers individuales a objetivos difíciles que sólo se pueden lograr con la cooperación sostenida. Con el proyecto de fetchmail he demostrado (en una escala mucho menor, claro) que sus métodos pueden copiarse con buenos resultados.

Posiblemente, lo mío fue realizado de una forma un poco más consciente y sistemática que la de él.

Muchas personas (especialmente aquellas que desconfían políticamente del libre mercado) podrían esperar que una cultura de individuos egoístas que se dirigen solos sea fragmentaria, territorial, clandestina y hostil. Pero esta idea es claramente refutada, por (por poner un ejemplo) la asombrosa variedad, calidad y profundidad de la documentación de Linux. Se da por un hecho que los programadores odian la documentación: ¿cómo entonces los hackers de Linux generan tanta? Evidentemente, el libre mercado en "egoboo" de Linux funciona mejor para producir tal virtuosismo, que los departamentos de edición, masivamente subsidiados, de los productores comerciales de software.

Tanto el proyecto de fetchmail como el del kernel de Linux han demostrado que con el estimulo apropiado al ego de otros hackers, un desarrollador/coordinador fuerte puede usar la Internet para aprovechar los beneficios de contar con un gran número de co-desarrolladores, sin que se corra el peligro de desbocar el proyecto en un auténtico relajo. Por lo tanto, a la Ley de Brooks yo le contrapongo lo siguiente:

19. Si el coordinador de desarrollo tiene un medio al menos tan bueno como lo es Internet, y sabe dirigir sin coerción, muchas cabezas serán, inevitablemente, mejor que una.

Pienso que el futuro del software libre será cada vez más de la gente que sabe como jugar el juego de Linus, la gente que deja atrás la catedral y abraza el bazar. Esto no quiere decir que la visión y la brillantez individuales ya no importen; al contrario, creo que en la vanguardia del software libre

estarán quienes comiencen con visión y brillantez individual, y luego las enriquezcan construyendo positivamente comunidades voluntarias de interés.

A lo mejor éste no sólo es el futuro del software libre. Ningún desarrollador comercial sería capaz de reunir el talento que la comunidad de Linux es capaz de invertir en un problema. ¡Muy pocos podrían pagar tan solo la contratación de las más de doscientas personas que han contribuido al fetchmail!

Es posible que a largo plazo triunfe la cultura del software libre, no porque la cooperación es moralmente correcta o porque la "apropiación" del software es moralmente incorrecta (suponiendo que se crea realmente en esto último, lo cual no es cierto ni para Linus ni para mí), sino simplemente por que el mundo comercial no puede ganar una carrera de armamentos evolutiva a las comunidades de software libre, que pueden poner mayores órdenes de magnitud de tiempo calificado en un problema que cualquier compañía.

Reconocimientos

Este artículo fue mejorado gracias a las conversaciones con un gran número de personas que me ayudaron a encontrar los errores. En especial, agradezco a Jeff Dutky dutky@wam.umd.edu, quien sugirió el planteamiento de que "la búsqueda de bugs pude hacerse en paralelo" y ayudó a ampliar el análisis respectivo. También agradezco a Nancy Lebovitz nancyl@universe.digex.net por su sugerencia de emular a Weinberg al imitar a Kropotkin. También recibí críticas perspicaces de Joan Eslinger wombat@kilimanjaro.engr.sgi.com y de Marty Franz marty@net-link.net de la lista de Técnicos Generales. Paul Egger eggert@twinsun.com me hizo ver el conflicto entre el GPL y el modelo de bazar. Estoy agradecido con los integrantes de PLUG, el Grupo de Usuarios de Linux de Filadelfia, por convertirse en el primer público para la primera versión de este artículo. Finalmente, los comentarios de Linus Torvalds fueron de mucha ayuda, y su apoyo inicial fue muy estimulante.

Otras lecturas

He citado varias partes del clásico de Frederick P. Brooks The Mythical Man- Month debido a que en muchos aspectos, todavía se tienen que mejorar sus puntos de vista. Yo recomiendo con cariño la edición del 25 aniversario de la Addison-Wesley (ISBN 0-201-83595-9), que viene junto a su artículo titulado Ninguna bala de plata.

La nueva edición trae una invaluable retrospectiva de veinte años, en la que Brooks admite francamente ciertas críticas al texto original que no pudieron mantenerse con el tiempo. Leí por primera vez la retrospectiva después de que estaba esencialmente terminado este artículo, y ¡me sorprendí al encontrar que Brooks le atribuye a Microsoft prácticas semejantes a las del bazar!

La psicología de la programación de computadoras de Gerald P. Wienberg (Nueva York, Van Nostrand Reinhold, 1971) introdujo el concepto infortunadamente denotado de "programación sin ego". A pesar de que él estaba muy lejos de ser la primera persona en comprender la futilidad del "principio de orden" fue probablemente el primero en reconocer y argumentar el tema en relación con el desarrollo del software.

Richard P. Gabriel, al analizar la cultura de UNIX anterior a la era de Linux, planteaba la superioridad de un primitivo modelo estilo bazar en un artículo de 1989: Lisp: Buenas noticias, malas noticias y cómo ganar en grande. Pese a estar atrasado en algunos aspectos, este ensayo es considerado correcto en algo por los admiradores de Lisp (entre quienes me incluyo). Uno de ellos me recordó que la sección titulada Lo peor es mejor predice con gran exactitud a Linux. Este artículo está disponible en la WWW en http://alpha- bits.ai.mit.edu/articles/good-news/good-news.html.

El trabajo de De Marco y Lister, Peopleware: Productive Projects and Teams (Nueva York; Dorset House, 1987; ISBN 0-932633-05-6) es una joya que ha sido subestimada; fue citada, para mi fortuna, por Fred Brooks en su retrospectiva. A pesar de que poco de lo que dicen los autores es directamente aplicable a las comunidades de software libre o de Linux, su visión sobre las condiciones necesarias para un trabajo creativo es aguda y muy recomendable para quien intente llevar algunas de las virtudes del modelo bazar a un contexto más comercial. Este documento esta disponible en http://www.agorics.com/agorpapers.htm

---

Epílogo: ¡Netscape adopta el bazar!

Es un extraño sentimiento el que se percibe cuando uno comprende que está ayudando a escribir historia…

El 22 de enero de 1998, aproximadamente siete meses después de que publiqué este artículo, Netscape Communications, Inc. anunció planes para liberar la fuente del Netscape Communicator. No tenía idea alguna de que esto iba a suceder antes de la fecha de anuncio.

Eric Hahn, Vicepresidente Ejecutivo y Director en Jefe de Tecnología en Netscape, me mandó un correo electrónico poco después del anuncio, que dice textualmente: "De parte de todos los que integran Netscape, quiero agradecerle por habernos ayudado a llegar hasta este punto, en primer lugar. Su pensamiento y sus escritos fueron inspiraciones fundamentales en nuestra decisión".

La siguiente semana, realicé un viaje en avión a Silicon Valley como parte de la invitación para realizar una conferencia de todo un día acerca de estrategia (el 4 de febrero de 1998) con algunos de sus técnicos y ejecutivos de mayor nivel. Juntos, diseñamos la estrategia de publicación de la fuente de Netscape y la licencia, y realizamos algunos otros planes que esperamos que eventualmente tengan implicaciones positivas de largo alcance sobre la comunidad de software gratuito. En el momento que estoy escribiendo, es demasiado pronto para ser más específico, pero se van a ir publicando los detalles en las semanas por venir.

Netscape está a punto de proporcionarnos con una prueba a gran escala, en el mundo real, del modelo del bazar dentro del ámbito empresarial. La cultura del software gratuito ahora enfrenta un peligro; si no funcionan las acciones de Netscape, entonces el concepto de software gratuito puede llegar a desacreditarse de tal manera que el mundo empresarial no lo abordará nuevamente sino hasta en una década.

Por otro lado, esto es también una oportunidad espectacular. La reacción inicial hacia este movimiento en Wall Street y en otros lados fue cautelosamente positiva. Nos están proporcionando una oportunidad de demostrar que nosotros podemos hacerlo. Si Netscape recupera una parte

significativa del mercado mediante este movimiento, puede desencadenar una revolución ya muy retrasada en la industria del software.

El siguiente año deberá demostrar ser un período muy interesante y de intenso aprendizaje.

Anexo: Lecciones enumeradas en La catedral y el bazar

1. Todo buen trabajo de software comienza a partir de las necesidades personales del programador (todo buen trabajo empieza cuando uno tiene que rascarse su propia comezón).

2. Los buenos programadores saben qué escribir. Los mejores, qué reescribir (y reutilizar).

3. "Considere desecharlo; de todos modos tendrá que hacerlo". (Fred Brooks, The Mythical Man-Month, Capítulo 11)

4. Si tienes la actitud adecuada, problemas interesantes te encontrarán.

5. Cuando pierdes el interés en un programa, debes darlo en herencia a un sucesor competente.

6. Tratar a los usuarios como colaboradores es la forma más apropiada de mejorar el código, y la más efectiva de depurarlo.

7. Libere rápido y a menudo, y escuche a sus clientes.

8. Dada una base suficiente de desarrolladores asistentes y beta-testers, casi cualquier problema puede ser caracterizado rápidamente, y su solución ser obvia al menos para alguien. O, dicho de manera menos formal, "con muchas miradas, todos los bugs saltarán a la vista". A esto lo he bautizado como la Ley de Linus.

9. Las estructuras de datos inteligentes y el código burdo funcionan mucho mejor que en el caso inverso.

10. Si usted trata a sus analistas (beta-testers) como si fueran su recurso más valioso, ellos le responderán convirtiéndose en su recurso más valioso.

11. Lo mejor después de tener buenas ideas es reconocer las buenas ideas de sus usuarios. Esto último es a veces lo mejor.

12. Con frecuencia, las soluciones más innovadoras y espectaculares provienen de comprender que la concepción del problema era errónea.

13. "La perfección (en diseño) se alcanza no cuando ya no hay nada que agregar, sino cuando ya no hay nada que quitar".

14. Toda herramienta es útil empleándose de la forma prevista, pero una *gran* herramienta es la que se presta a ser utilizada de la manera menos esperada.

15. Cuándo se escribe software para una puerta de enlace de cualquier tipo, hay que tomar la precaución de alterar el flujo de datos lo menos posible, y ¡*nunca* eliminar información a menos que los receptores obliguen a hacerlo!

16. Cuando su lenguaje está lejos de un Turing completo, entonces el azúcar sintáctico puede ser su amigo.

17. Un sistema de seguridad es tan seguro como secreto. Cuídese de los secretos a medias.

18. Para resolver un problema interesante, comience por encontrar un problema que le resulte interesante.

19. Si el coordinador de desarrollo tiene un medio al menos tan bueno como lo es Internet, y sabe dirigir sin coerción, muchas cabezas serán, inevitablemente, mejor que una.